Lwt: a Cooperative Thread Library

Jérôme Vouillon

Laboratoire Preuves, Programmes et Systèmes
Université Paris Diderot (Paris 7), CNRS
Paris, France
jerome.vouillon@pps.jussieu.fr

Abstract

We present a cooperative thread library for Objective Caml. The library is entirely written in Objective Caml and does not rely on any external C function. Programs involving threads are written in a monadic style. This makes it possible to write threaded code almost as regular ML code, even though it has a different semantics. Cooperative threads are especially well suited for concurrent network applications, where threads perform little computation and spend most of their time waiting for input or output, at which time other threads can run. This library has been successfully used in the Unison file synchronizer and the Ocsigen Web server.

Categories and Subject Descriptors D.1.1 [*PROGRAMMING TECHNIQUES*]: Applicative (Functional) Programming; D.1.3 [*PROGRAMMING TECHNIQUES*]: Concurrent Programming; D.3.3 [*PROGRAMMING LANGUAGES*]: Language Constructs and FeaturesConcurrent programming structures

General Terms Design, Languages

Keywords Thread, Concurrency, Networking, Programming, Implementation, Monad, Objective Caml, ML

1. Introduction

The `Lwt` ("lightweight threads") library is a cooperative thread library for Objective Caml (Leroy et al.): threads have to explicitly yield the control to other threads rather than being preempted by a scheduler. The motivations for this library are two-fold. First, we believe that cooperative threads are easier to program with than preemptive threads. Indeed, as context switches are explicit (typically through a call to a `yield` function or due to a blocking system call), there are much less opportunities for race conditions. Second, compared to system threads, the threads are extremely lightweight: they only hold the values required for the remaining of the computation rather than a full stack. Therefore, there can easily be thousands of them. Context switch is also cheaper as it consists in just a few function calls. Cooperative threads are especially well suited for highly concurrent network applications (von Behren et al. 2003), where threads perform little computation and spend most of their time waiting for input or output, at which time other threads can run.

The library is entirely written in Objective Caml, contrary to the standard thread libraries (system threads and VM-level threads) which rely on external C functions. Programs involving threads are written in a monadic style. This makes it possible to write threaded code almost as regular ML code, even though it has a different semantics.

This library has been successfully used since 2002 in the Unison file synchronizer (Balasubramaniam and Pierce 1998; Pierce and Vouillon 2004) for the simultaneous synchronization of several files. It is now used in the Ocsigen Web server (Balat 2006). The library is available online[1]. It is also included in the Debian Linux distribution.

We start with simple examples giving a feel of the library (section 2). We detail the library API (section 3). We provide more complex examples demonstrating how to use the library (section 4). Finally, we describe the implementation (section 5).

2. A quick presentation of the library

In order to give a feel of the library, we present two simple examples. With the first example, we explain how to write a function performing I/Os asynchronously. The second one shows two threads running concurrently.

First, consider the following function. It reads up to one kilobyte from file descriptor `in_fd`, sleeps for three seconds and writes the data to file descriptor `out_fd`. Any of the three system calls may block, interrupting for some time the whole program.

```
let forward in_fd out_fd buffer =
  let n = Unix.read in_fd buffer 0 1024 in
  Unix.sleep 3;
  let n' = Unix.write out_fd buffer 0 n in
  ()
```

The core idea of the library is to replace a possibly blocking function by a function that returns immediately a *promise*, that is a value that acts as a proxy for the value eventually computed by the function. Thus, instead of function `Unix.sleep`, one uses function `Lwt_unix.sleep` of type `int -> unit Lwt.t` where `Lwt.t` is the type of promises. Likewise, the function `Unix.read` is replaced by a function `Lwt_unix.read` of type

```
file_descr -> string -> int -> int -> int Lwt.t
```

Type `Lwt.t` can also be interpreted as the type of *threads*: the function `Lwt_unix.read` returns a thread that reads from a given file descriptor and terminates with return value the number of byte read. We adopt this interpretation for the remainder of the paper.

The library provides basic functions for handling threads. The function `Lwt.bind` is used to perform an action when the return value of a thread becomes available: the expression

```
Lwt.bind e (fun x -> e')
```

evaluates to a thread `t` that first waits for thread `e` to terminates. The return value is then bound to parameter `x` in order to execute thread `e'`. The thread `t` behaves thereafter as thread `e'`. Intuitively, this can be read as a `let` expression for threads: "`let x = e in e'`". It is also a synchronization primitive: thread `e'` starts running only after thread `e` is terminated. The thread `Lwt.return e` is a thread that terminates immediately with value

[1] `http://www.pps.jussieu.fr/~vouillon/lwt/`

the value of expression `e`. The type of threads together with functions `Lwt.bind` and `Lwt.return` forms a monad.

In order to use the library, the function `forward` is rewritten as follows:

```
let forward in_fd out_fd buffer =
  Lwt.bind
    (Lwt_unix.read in_fd buffer 0 1024)
    (fun n ->
       Lwt.bind
         (Lwt_unix.sleep 3)
         (fun () ->
            Lwt.bind
              (Lwt_unix.write out_fd buffer 0 n)
              (fun n' ->
                 Lwt.return ())))
```

Thus, the function now returns immediately (that is, without blocking in a system call) a thread of type `unit Lwt.t`. This thread first reads up to one kilobyte. When `n` bytes have been read, the thread sleeps for three seconds. Then, it writes the `n` bytes. Finally, it returns the unit value.

The syntax can be made more readable by using a binary operator for `Lwt.bind`:

```
let (>>=) = Lwt.bind
```

Then, the function can be rewritten as follows.

```
let forward in_fd out_fd buffer =
  Lwt_unix.read in_fd buffer 0 1024 >>= fun n ->
  Lwt_unix.sleep 3 >>= fun () ->
  Lwt_unix.write out_fd buffer 0 n >>= fun n' ->
  Lwt.return ()
```

The second example is self-contained. It presents the thread scheduler. We first define a function `loop`. This function returns a thread printing a text each second.

```
let rec loop s =
  Lwt_unix.sleep 1 >>= fun () ->
  Format.printf "Hello %s@." s;
  loop s
```

Next, two threads are created. This is done implicitly: threads starts executing at once and there is no need to call a function such as `spawn` to launch them. The threads are immediately suspended as they call `Lwt_unix.sleep`. Then, we enter the scheduler, which takes care of resuming the appropriate threads when a system call terminates: the expression `Lwt_unix.run e` executes the scheduler until thread `e` terminates. Here, the scheduler runs for ever as the thread `Lwt.wait ()` is a thread that never terminates.

```
let _ =
  let thread1 = loop "A" in
  let thread2 = loop "B" in
  Lwt_unix.run (Lwt.wait ())
```

3. The library API

The library has two main components, which we present in turn. The module `Lwt` provides the core functionalities. The module `Lwt_unix` provides a scheduler and gives access to the Unix I/O system calls. Some additional modules implement functionalities such as locks ant timeouts.

3.1 The core library

The interface of module `Lwt` is given in figure 1. The type `'a t` is the type of threads which, if they terminate successfully, yield a value of type `'a`. A thread can also run for ever, or fail with some exception.

```
type 'a t

val return : 'a -> 'a t
val bind : 'a t -> ('a -> 'b t) -> 'b t
val fail : exn -> 'a t
val catch :
  (unit -> 'a t) -> (exn -> 'a t) -> 'a t
val try_bind :
  (unit -> 'a t) ->
  ('a -> 'b t) -> (exn -> 'b t) -> 'b t

val wait : unit -> 'a t
val wakeup : 'a t -> 'a -> unit
val wakeup_exn : 'a t -> exn -> unit
val poll : 'a t -> 'a option
```

Figure 1. The signature of module `Lwt`

The thread `return e` is a thread that terminates immediately with return value the value of expression `e`. The thread `bind t f` is a thread which first waits for the completion of thread `t` and then, if the thread succeeds, behaves as the application of function `f` to the return value of thread `t`. If the thread `t` fails, `bind t f` also fails, with the same exception. As mentioned in section 2, it is convenient to use the binary operator >>= as an alias for function `bind`.

The type `t` together with function `bind` and `return` forms a monad. The following laws are satisfied:

$$\texttt{return e >>= fun x -> t} \quad \equiv \quad \texttt{let x = e in t}$$

$$\texttt{t >>= fun x -> return x} \quad \equiv \quad \texttt{t}$$

$$\begin{gathered}\texttt{(t >>= fun x -> t') >>= fun y -> t''} \\ \equiv \\ \texttt{t >>= fun x -> (t' >>= fun y -> t'')}\end{gathered}$$

These are the monad laws adapted to the effectful semantics of ML.

The thread `fail e` is a thread that fails immediately with exception `e`. The thread `catch (fun () -> t) f` behaves as thread `t` as long as this thread does not fail. If the thread fails with some exception `e`, it behaves thereafter as the thread resulting from applying function `f` to exception `e`. Functions `fail` and `catch` satisfy the following laws:

$$\texttt{fail e >>= fun x -> t} \quad \equiv \quad \texttt{fail e}$$

$$\begin{gathered}\texttt{catch (fun () -> fail e) (fun x -> t)} \\ \equiv \\ \texttt{let x = e in t}\end{gathered}$$

$$\begin{gathered}\texttt{catch (fun () -> return e) (fun x -> t)} \\ \equiv \\ \texttt{return e}\end{gathered}$$

For the sake of convenience, the function `catch` also captures exceptions raised by the usual ML exception mechanism. For instance, in the following expression, the exception `Exit` is caught and applied to function `h`.

```
catch (fun () -> raise Exit) h
```

The function `try_bind` is a generalization of functions `bind` and `catch`, as proposed in (Benton and Kennedy 2001). The thread `try_bind (fun () -> t) f g` behaves as `bind t f` if thread `t` does not fail (and no exception is raised during its evaluation). It behaves as `catch t g` otherwise.

The last four functions are less useful for the casual user. They provide the basic primitives on which can be built multi-threaded libraries such as `Lwt_unix`. The thread `wait ()` is a thread which is suspended, possibly forever. The function `wakeup` terminates such a thread with a given return value. The function `wakeup_exn` makes it fail with a given exception. The expression `poll t` tests the state of thread `t`. It returns `Some v` if thread `t` is terminated and returned the value `v`. If the thread failed with some exception, this exception is raised. If the thread is still running, the value `None` is returned without blocking.

3.2 The Lwt Unix module

This module provides all functionalities required to perform Unix I/O system calls asynchronously. The module implements its own type of file descriptors. This ensures that all file descriptors used in the library are in non-blocking mode, which is mandatory for the library to work properly. The library also ensures that, once a descriptor is closed, any attempt to access it fails. This is not the case when using a standard Unix library, where a same descriptor can be reused for another file. This provides a protection against programming errors with thread continuing to write to a closed descriptor and ending up writing to the wrong file. Conversion functions from and to the file descriptors of the standard `Unix` library are provided.

The module exports the usual I/O functions such as `read`, `write` or `connect` corresponding to the ones in the `Unix` library. The types of functions that do not block such as `close` or `listen` are unchanged:

```
val close : file_descr -> unit
```

Possibly blocking functions have their types changed to return a thread:

```
val read : file_descr ->
           string -> int -> int -> int Lwt.t
```

The module also implements a scheduler. The role of the scheduler is to repeatedly wait for events (timeouts, file descriptors ready for read or write, ...) and to resume the appropriate threads. The scheduler is entered by calling function `run` of type

```
'a Lwt.t -> 'a.
```

The expression `run t` executes thread `t` (and other concurrent threads) until it terminates. The expression evaluates to the return value of the thread. If the thread fails with some exception, this exception is raised. A thread can suspend itself temporarily to let other threads run by calling function `yield`.

Finally, the function `abort` of type

```
file_descr -> exn -> unit
```

aborts all current and future operations on the given file descriptor with the given exception (except for function `close` so that the file descriptor can be properly closed). This provides a convenient way to interrupt a thread that loops reading on or writing to a given file descriptor, for instance after some timeout.

3.3 Additional modules

The following modules are also available:

- the module `Lwt_chan` provides buffered I/O functions, similar to the ones in module `Pervasive` of the Objective Caml standard library;
- the module `Lwt_mutex` implements mutual exclusion locks (see section 4.3 for details);
- the module `Lwt_timeout` provides a timeout mechanism; it can be used for instance with `Lwt_unix.abort` to close a network connection after some idle time.
- the module `Lwt_preemptive` allows to mix preemptive threads with `Lwt` cooperative threads. It maintains an extensible pool of preemptive threads on with computations can be performed.

3.4 Pitfalls

While using the library, we have noticed a number of pitfalls that we document here.

First, it is mandatory to use the functions `catch` or `try_bind` for handling thread exceptions rather than the usual exception handling mechanism: an expression

```
try t with e -> raise Some_Exception
```

will not catch any exception embedded in thread `t`. This is a point that one must keep in mind when converting existing code to use the library. Indeed, the type checker properly force us to use `bind` and `return` wherever needed but the expression above remains well-typed whatever the type of expression `t`.

Second, the programmer expects uncaught exceptions to terminate the program. This is not the case with an exception embedded in a thread. For instance, the expression `fail e` has no direct effect. Thus, an exception handler should always be explicitly wrapped around any thread that may fail with an exception, so that the exception is not silently ignored.

Last, it is very tempting to call function `Lwt_unix.run` to get the result of a thread without using function `bind` when not in a thread context, given the type of this function:

```
'a Lwt.t -> 'a.
```

This works properly only as long as the call is not made from a thread. Calling this function from a thread may result in a deadlock, as it introduces spurious synchronizations between threads. Indeed, call to function `run` are regular function calls and thus must be properly nested. Thus, if two threads call function `run`, the first thread cannot exit from the function before the second one.

4. Using the library

The following three larger examples illustrate the use of the library. The first one, a port forwarder (section 4.1), is a typical example of network programming. The two other ones, a simple scheduler (section 4.2) and an implementation of locks (section 4.3), demonstrate the expressivity of the short API of module `Lwt`.

4.1 Network programming

We present the implementation of a port forwarder. The program waits for connections on port 8080. When a client connects to this port, a remote connection to port 80 on `google.com` is established. Any data received on either side is then forwarded to the other side.

The auxiliary function `really_write` write `l` bytes of string `s` starting at position `p` to file descriptor `o`. Several calls to function `Lwt_unix.write` may be needed as the system call may write less than `l` bytes.

```
let rec really_write o s p l =
  Lwt_unix.write o s p l >>= fun n ->
  if l = n then
    Lwt.return ()
  else
    really_write o s (p + n) (l - n)
```

The function `forward` writes everything it reads on file descriptor `i` to file descriptor `o`. When the `Lwt_unix.read` system call returns zero, which indicates end of file, we shut down the sending part of the other connection and stop transmitting.

```
let rec forward i o =
  let s = String.create 1024 in
```

```
    Lwt_unix.read i s 0 1024 >>= fun l ->
    if l > 0 then begin
      really_write o s 0 l >>= fun () ->
      forward i o
    end else begin
      Lwt_unix.shutdown o Unix.SHUTDOWN_SEND;
      Lwt.return ()
    end
```

The function `new_socket` creates a TCP socket. The program expects connections on address `local_addr` and establish connections to address `remote_addr`.

```
  let new_socket () =
    Lwt_unix.socket
      Unix.PF_INET Unix.SOCK_STREAM 0
  let local_addr =
    Unix.ADDR_INET (Unix.inet_addr_any, 8080)
  let remote_addr =
    let host_entry =
      Unix.gethostbyname "google.com" in
    let inet_addr =
      host_entry.Unix.h_addr_list.(0) in
    Unix.ADDR_INET (inet_addr, 80)
```

The function `accept` returns a thread that repeatedly accept connections on the socket `sock`. The thread blocks until a connection is established. Then, a thread dealing with the connection is started asynchronously (parenthesized expression) and the function is called recursively to wait for another connection. The thread connects to the remote address. It starts two threads for forwarding data in both directions. It then waits for the two threads to terminate and finally closes the file descriptors corresponding to both connections.

```
  let rec accept sock =
    Lwt_unix.accept sock >>= fun (inp, _) ->
    ignore
      (let out = new_socket () in
       Lwt_unix.connect
         out remote_addr >>= fun () ->
       let io = forward inp out in
       let oi = forward out inp in
       io >>= fun () -> oi >>= fun () ->
       Lwt_unix.close out;
       Lwt_unix.close inp;
       Lwt.return ());
    accept sock
```

We can now write the body of the program: it creates a socket that listens on the local address and starts accepting connections.

```
  let _ =
    let socket = new_socket () in
    Lwt_unix.setsockopt
      socket Unix.SO_REUSEADDR true;
    Lwt_unix.bind socket local_addr;
    Lwt_unix.listen socket 1024;
    Lwt_unix.run (accept socket)
```

Clearly, a robust version of this program should catch Unix errors so as to properly shutdown the connections. Besides, writing to a socket whose other end is closed results by default in a `SIGPIPE` signal. This signal should be ignored by the program so that `Lwt_unix.write` fails with an `EPIPE` error instead.

4.2 A simple scheduler

We present a simple scheduler. The implementation of the more complete scheduler provided by module `Lwt_unix` is sketched in section 5.6. Here, a thread can temporarily pause by calling a function `yield`, in order to allow other threads to execute. The task of this scheduler is to then restart another thread. The scheduler is started by calling a function `run`.

We first define the FIFO queue of suspended thread, using the `Queue` module of Objective Caml standard library.

```
  let queue = Queue.create ()
```

The scheduler repeatedly takes a thread from the queue and resumes it. The thread runs until a call to `yield`, which gives back the control to the scheduler which can then restart another thread. The scheduler stops when the queue becomes empty.

```
  let rec run () =
    match
      try
        Some (Queue.take queue)
      with Queue.Empty ->
        None
    with
      None   -> ()
    | Some t -> Lwt.wakeup t (); run ()
```

The function `yield` creates a suspended thread, adds the thread to the end of the queue and returns it. Thus, a thread waiting for an expression `yield ()` to terminate is stopped until resumed by the scheduler.

```
  let yield () =
    let res = Lwt.wait () in
    Queue.push res queue;
    res
```

Here is an example of use of the scheduler. The `loop` function prints `n` times a string `s`, letting other threads run at each iteration.

```
  let rec loop s n =
    if n > 0 then begin
      Format.printf "%s@." s;
      yield () >>= fun () ->
      loop s (n - 1)
    end else
      Lwt.return ()
```

Two threads are started, that output alternatively `"a"` and `"b"`.

```
  let _ =
    let ta = loop "a" 6 in
    let tb = loop "b" 5 in
    run ()
```

4.3 Mutexes

We present an implementation of mutual exclusion locks. It follows the corresponding implementation in the Objective Caml standard thread library. Interestingly, this implementation only depends on module `Lwt`. It can thus be used with any scheduler. A mutex is a pair of a boolean which indicates whether the mutex is locked and a list of threads waiting for the mutex to become unlocked.

```
  type t =
     { mutable locked : bool;
       mutable waiting : unit Lwt.t list  }
```

A mutex is initially created unlocked.

```
  let create () =
    { locked = false; waiting = [] }
```

The function `lock` attempts to lock a mutex `m`. If the mutex is not locked, it is locked and the function returns immediately. Otherwise, the thread is suspended: a suspended thread is created and added to the list of waiting threads. The function waits for this thread to be resumed and then calls itself recursively.

```
let rec lock m =
  if not m.locked then begin
    m.locked <- true;
    Lwt.return ()
  end else begin
    let res = Lwt.wait () in
    m.waiting <- res :: m.waiting;
    res >>= fun () ->
    lock m
  end
```

In order to release a mutex, a copy of the set of waiting threads `w` is extracted and the set is cleared. The mutex is then unlocked. Finally, the waiting threads are restarted. It is important to perform this step last in order to avoid race conditions.

```
let unlock m =
  let w = m.waiting in
  m.waiting <- [];
  m.locked <- false;
  List.iter (fun t -> Lwt.wakeup t ()) w
```

5. Implementation

We first present in details the core module `Lwt`. The implementation of module `Lwt_unix` is then sketched in section 5.6. The code presented here deviates slightly from the actual implementation for the sake of readability.

5.1 The type of threads

Threads are represented by a memory cell with a mutable state.

```
type 'a t =
  { mutable state : 'a state }
and 'a state =
    Return of 'a
  | Fail of exn
  | Sleep of ('a t -> unit) list ref
  | Link of 'a t
```

The three main states are:

- `Return v`: the thread has terminated successfully with the value `v`;
- `Fail e`: the thread has failed with exception `e`;
- `Sleep w`: the thread is not finished yet; the thunk functions in set `w` are called when the thread terminates.

The last state `Link t` is used to implement a union-find datastructure[2] over threads in order to coalesce threads with an identical behavior. This turns out to be crucial to avoid some memory leaks (see section 5.5.2 for details). The `find` function returns the representative of a thread.

```
let rec find t =
  match t.state with
    Link t' -> find t'
  | _       -> t
```

The actual implementation of function `find` uses path compression: each visited thread gets directly linked to its representative.

5.2 Creating a thread

There is a function for creating a thread in each state.

```
let return v = { state = Return v }
let fail e   = { state = Fail e }
let wait ()  = { state = Sleep (ref []) }
```

5.3 Terminating a thread

We present the implementation of the two functions `wakeup` and `wakeup_exn`. They both rely on a function `finish` that changes the state of a thread from still running to terminated. Its arguments are the thread `t` and the new state `st` (which should be either `Return v` or `Fail e`). First, the representative of the thread is found. Then, the list of waiters is extracted and the state of the thread is changed to `st`. Finally, the waiters are awaken. It is crucial to perform this step last in order to avoid a race condition where new waiters are added while processing current waiters.

```
let finish t st =
  let t = find t in
  match t.state with
    Sleep waiters ->
      t.state <- st;
      List.iter (fun f -> f t) !waiters
  | _ ->
     invalid_arg "finish"
```

The implementation of functions `wakeup` and `wakeup_exn` is now straightforward.

```
let wakeup t v     = finish t (Return v)
let wakeup_exn t e = finish t (Fail e)
```

5.4 Thread synchronization

We present the implementation of functions `bind`, `try_bind` and `catch`. These functions make use of a function `trap` for catching ML exceptions and embed them into a failing thread.

```
let trap f x = try f x with e -> fail e
```

They also rely on a function `connect` of type:

`'a t -> 'a t -> unit.`

A call `connect t t'`, where thread `t` must not be terminated, ensures that the behavior of thread `t` mimics thereafter the behavior of thread `t'`: thread `t` will terminate when thread `t'` terminates, with the same result. The easy case is when `t'` is already finished. Then, thread `t` is terminated with the same state as `t'` (call to function `finish`). Otherwise, the threads are both still running. Then, the representative of `t'` is linked to the representative of `t` and the waiter sets are merged. The actual implementation uses lists with constant-time append in order to make the cost of this last operation independent from the number of waiters.

```
let rec connect t t' =
  let t' = find t' in
  match t'.state with
    Sleep waiters' ->
      let t = find t in
      begin match t.state with
        Sleep waiters ->
          waiters := !waiters' @ !waiters;
          t'.state <- Link t
      | _ ->
          invalid_arg "connect"
      end
  | _ ->
      finish t t'.state
```

All three synchronization functions share a common core, function `try_bind_rec`. This function takes as argument a thread `t` and two functions `f` and `g`. If `t` is terminated with value `v`, then the

[2] http://en.wikipedia.org/wiki/Union-find

application of function f to value v is returned. If t has failed with exception e, then the application of function g to exception e is returned. If t is not yet terminated, a fresh suspended thread res is created. A thunk is added to the set of waiters of thread t so that, when thread t terminates, the function try_bind_rec is called again and the behavior of res follows the behavior of the thread returned by this function. The thread res is finally returned. The function try_bind_rec is called recursively at most once, as t is always terminated when the recursive call is performed.

```
let rec try_bind_rec t f g =
  match (find t).state with
    Return v ->
      f v
  | Fail e ->
      g e
  | Sleep waiters ->
      let res = wait () in
      waiters :=
       (fun t ->
          connect res
            (try_bind_rec t (trap f) (trap g)))
          ::
       !waiters;
      res
  | Link _ ->
      assert false
```

From this function, all three synchronization operators can easily be written:

```
let bind t f = try_bind_rec t f fail

let try_bind f g h =
  try_bind_rec (trap f ()) g h

let catch f g = try_bind f return g
```

The use of function trap in the functions try_bind_rec and try_bind is clarified in section 5.5.1

5.4.1 Polling for a thread state

The implementation of the function testing the state of a thread is straightforward.

```
let poll t =
  match (repr t).state with
    Fail e   -> raise e
  | Return v -> Some v
  | Sleep    -> None
  | Repr _   -> assert false
```

5.5 Implementation difficulties

The implementation has now been presented in full. The following explains in more details some subtle issues.

5.5.1 Dealing with exceptions

The function Lwt.catch as used in the expression below will catch not just the exceptions embedded in thread t (for instance, using the function fail) but also the exceptions raised by the usual ML mechanism (using the operator raise).

```
Lwt.catch (fun () -> t) handler
```

This is implemented by calling function trap at suitable places. When the exception is raised during the evaluation of the thunk, it is caught by the occurrence of this function in the body of function try_bind. This covers the following example:

```
Lwt.catch
  (fun () ->
     raise Not_found;
     Lwt.return ())
  handler
```

The following second case, where function Lwt.bind is applied to a thread which is already terminated, is also covered by the same call to function trap. Indeed, the exception is raised before the whole thunk function exits.

```
Lwt.catch
  (fun () ->
     Lwt.return () >>= fun () ->
     raise Not_found;
     Lwt.return ())
  handler
```

In the last case below, the evaluation of the thunk results in a suspended thread, and the exception is raised only after the thread resumes. In this case, the exception is caught by the calls to function trap in function try_bind_rec.

```
Lwt.catch
  (fun () ->
     Lwt_unix.yield () >>= fun () ->
     raise Not_found;
     Lwt.return ())
  handler
```

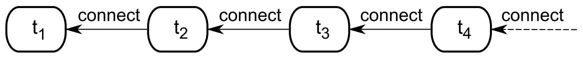
One has to be careful about where the calls to function trap are performed. In particular, it would be incorrect to protect systematically the second argument of function bind, as this would break tail-recursion. However, it is not an issue to do it after a recursive call to function try_bind_rec, that is, in response to a thread termination. Indeed, this termination is generally performed by a call to function wakeup or function wakeup_exn from the scheduler, and the scheduler body is a tail-recursive function and is thus at a fixed stack depth.

5.5.2 Avoiding memory leaks

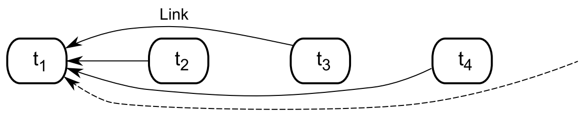
It is crucial that the memory behavior of threads conforms to the expectations of the programmer. In particular, a function which is tail-recursive when written in a non-threaded way should be tail-recursive when using Lwt. This is not the case with a naive implementation that does not coalesce equivalent threads. Indeed, consider the following piece of code, using the scheduler in section 4.2.

```
let rec loop n =
  if n = 0 then Lwt.return () else
  yield () >>= fun () -> loop (n - 1)
in
let l = loop 100000 in
run ()
```

The function yield returns a suspended thread t. The function loop thus also initially returns a suspended thread t_1. Later, the thread t is successfully terminated by the scheduler. This triggers a recursive call to function loop returning a new thread t_2 which is connected to thread t_1 by function connect. This is repeated again and again, thus we get a longer and longer chain:



A straightforward implementation of function connect, when applied to threads t_i and t_{i+1}, would add a thunk to the list of waiters of thread t_{i+1}. The goal of this thunk is to update thread t_i when thread t_{i+1} is terminated. But then, as long as the head of the chain remains live (that is, as long as the loop has not ended), none of the chain can be garbage collected. Our implementation uses the fact that all these threads behave the same: they are suspended until the loop ends, and then are all terminated with value unit. It takes thread t_1 as the unique representative for them all. Then, all intermediate threads can be garbage collected:



We conjecture that, with this implementation, translating an existing code to use Lwt does not introduce any memory leak.

5.6 The Unix library

We sketch the implementation of the scheduler and present the implementation of a possibly blocking system call. For the sake of clarity, we only present the way the scheduler deals with operations on file descriptors. The actual implementation also deals with threads to be restarted after a given amount of time (functions sleep and yield) and subprocesses termination (function waitpid).

The threads waiting for I/Os are stored in two datastructures inputs and outputs that associate to some file descriptors the actions that should be performed when they become available respectively for reading and writing.

The scheduler (function run below) loops until the input thread thr terminates. The status of the thread is checked at each iteration by calling function Lwt.poll. If it is not finished yet, the scheduler proceeds to call the function Unix.select to wait for file descriptors to become available. The lists infds and outfds of file descriptors to watch are computed by function active_descriptors. The third list is for waiting for so-called socket exceptions (that is, out-of-band data), which is not currently supported (this socket feature is hardly ever used). The float -1.0 indicates that the wait is unbounded (no timeout). The system call is interrupted with error EINTR when a signal occurs, resulting in an ML exception. This exception can be ignored. When the system call returns, the corresponding actions are performed on each available descriptor by calling function perform_action. Finally, the scheduler calls itself recursively.

```
let rec run thr =
  match Lwt.poll thr with
    Some v ->
      v
  | None ->
      let infds = active_descriptors inputs in
      let outfds = active_descriptors outputs in
      let (readers, writers, _) =
        try
          Unix.select infds outfds [] (-1.0)
        with
          Unix_error (Unix.EINTR, _, _) ->
            ([], [], [])
      in
      List.iter
        (fun fd -> perform_actions inputs fd)
        readers;
      List.iter
        (fun fd -> perform_actions outputs fd)
        writers;
      run thr
```

In the actual implementation, timeouts (function sleep) are managed by using a priority queue. A timeout value is given to function select instead of the float -1.0 in order to interrupt the system call when a thread has to be resumed. Child termination (function waitpid) is detected by catching the signal SIGCHILD.

The function perform_actions eventually invokes the function wrap_syscall shown below to perform an action action on file descriptor ch. In order to be able requeue the action in case it fails to complete, the function takes as additional arguments the datastructure set that held the action and the thread cont to be resumed when the action is completed. The function check_descriptor is called to check whether the function abort was previously called on the file descriptor and raises the corresponding exception if this is the case (see section 3.2). It also raises an exception when the file descriptor is marked as closed. The action is then attempted. If it fails with a Unix error EAGAIN, EWOULDBLOCK or EINTR, the action is requeued (function add_action). Otherwise, the thread is resumed.

```
let rec wrap_syscall set ch cont action =
  let res =
    try
      check_descriptor ch;
      Success (action ())
    with
      Unix.Unix_error
          ((Unix.EAGAIN | Unix.EWOULDBLOCK |
            Unix.EINTR),_,_) ->
        add_action set ch cont action;
        Requeued
    | e ->
        Exn e
  in
  match res with
    Success v ->
      Lwt.wakeup cont v
  | Exn e ->
      Lwt.wakeup_exn cont e
  | Requeued ->
      ()
```

We now show how the function write is implemented. It first checks the file descriptor. If this does not result in an exception, the system call write is performed. The Unix errors indicating that the write would block are caught and result in scheduling the write to be attempted again when the file descriptor becomes available. This is performed by calling function register_action which returns a suspended thread that is resumed when the action completes. If another error occurs, a failing thread is returned.

```
let write ch buf pos len =
  try
    check_descriptor ch;
    Lwt.return (Unix.write ch.fd buf pos len)
  with
    Unix.Unix_error
        ((Unix.EAGAIN | Unix.EWOULDBLOCK |
          Unix.EINTR), _, _) ->
      register_action outputs ch
        (fun () -> Unix.write ch.fd buf pos len)
  | e ->
      Lwt.fail e
```

6. Related work

The idea to implement cooperative threads using a monad is due to Claessen (Claessen 1999). Li and Zdancewic have written an implementation in Haskell (Li and Zdancewic 2007) with performance in mind. They use the efficient epoll Linux mechanism instead of the more portable but less efficient select system call to implement their scheduler. The company Liveops has developed a similar monad-based library (Waterson 2007) for Objective Caml. They report that their library does not deal well with exceptions. The library has not been publicly released yet.

These works all use some variants of a continuation monad, which makes the semantics of their threads slightly different from ours. The type of threads is typically similar to the following one:

```
'a t = ('a -> unit) -> unit
```

This is a functional type: an expression of this type does nothing before being given a continuation. For instance, an expression `read fd buffer 0 512` does not attempt at once to read on file descriptor `fd` as is the case with `Lwt`. The read is only attempted once a continuation is provided. Thread creation is then explicit: a function, usually called `fork` or `spawn`, must be called to apply a thread to its final continuation and thus start the thread execution. As an expression of type `'a t` is not a running thread, the function `bind` is just a sequencing operator and does not provide a communication mechanism between threads. A separate mechanism has to be provided.

An alternative to threads for highly concurrent network applications is event-based programming. The `equeue` library, part of the Ocamlnet (Stolpmann) library, is an Objective Caml library for event queues. It is used by a number of other libraries in Ocamlnet to parallelize network code. Compared to `Lwt`, the API is very low-level. It should be possible to built a `Lwt` scheduler on top of this library, so that code written using `Lwt` can interact with the library.

We have not performed any benchmark, but we believe that our implementation is competitive performance-wise with respect to other Objective Caml thread implementations. Indeed, they all share the same limitation that only one thread is active at a given time. An advantage of our library is the low cost of thread creation.

Acknowledgments

A first version of this library was written during a postdoctoral fellowship at the University of Pennsylvania. Benjamin Pierce and Zhe Yang contributed some documentation for this library. Vincent Balat encouraged me to expand the library.

References

S. Balasubramaniam and Benjamin C. Pierce. What is a file synchronizer? In *Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '98)*, October 1998. Full version available as Indiana University CSCI technical report #507, April 1998.

Vincent Balat. Ocsigen: Typing Web interaction with Objective Caml. In *International Workshop on ML*, pages 84–94. ACM Press, 2006. ISBN 1-59593-483-9. doi: http://doi.acm.org/10.1145/1159876.1159889.

Nick Benton and Andrew Kennedy. Exceptional syntax. *J. Funct. Program.*, 11(4):395–410, 2001. ISSN 0956-7968. doi: http://dx.doi.org/10.1017/S0956796801004099.

Koen Claessen. A poor man's concurrency monad. *J. Funct. Program.*, 9(3):313–323, 1999. ISSN 0956-7968. doi: http://dx.doi.org/10.1017/S0956796899003342.

Xavier Leroy, Damien Doligez, Jacques Garrigue, Jérôme Vouillon, and Dider Rémy. The Objective Caml system. Software and documentation available on the Web, `http://pauillac.inria.fr/ocaml/`.

Peng Li and Steve Zdancewic. Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 189–199, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. doi: http://doi.acm.org/10.1145/1250734.1250756.

Benjamin C. Pierce and Jérôme Vouillon. What's in Unison? A formal specification and reference implementation of a file synchronizer. Technical Report MS-CIS-03-36, Dept. of Computer and Information Science, University of Pennsylvania, 2004.

Gerd Stolpmann. The ocamlnet library. Software and documentation available on the Web, `http://projects.camlcity.org/projects/ocamlnet.html`.

Rob von Behren, Jeremy Condit, and Eric Brewer. Why events are a bad idea (for high-concurrency servers). In *HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems*, pages 4–4, Berkeley, CA, USA, 2003. USENIX Association.

Chris Waterson. An ocaml-based network services platform. In *CUFP '07: Proceedings of the 4th ACM SIGPLAN workshop on Commercial users of functional programming*, pages 1–2, New York, NY, USA, 2007. ACM. doi: http://doi.acm.org/10.1145/1362702.1362711.